Assessing asymmetric fault-tolerant software

Peter Popov, Lorenzo Strigini
Centre for Software Reliability
City University London
United Kingdom
ptp@csr.city.ac.uk, strigini@csr.city.ac.uk

***Abstract* — The most popular forms of fault tolerance against design faults use "asymmetric" architectures in which a "primary" part performs the computation and a "secondary" part is in charge of detecting errors and performing some kind of error processing and recovery. In contrast, the most studied forms of software fault tolerance are "symmetric" ones, e.g. N-version programming. The latter are often controversial, the former are not. We discuss how to assess the dependability gains achieved by these methods. Substantial difficulties have been shown to exist for symmetric schemes, but we show that the same difficulties affect asymmetric schemes. Indeed, the latter present somewhat subtler problems. In both cases, to predict the dependability of the fault-tolerant system it is not enough to know the dependability of the individual components. We extend to asymmetric architectures the style of probabilistic modeling that has been useful for describing the dependability of "symmetric" architectures, to highlight factors that complicate the assessment. In the light of these models, we finally discuss fault injection approaches to estimating coverage factors. We highlight the limits of what can be predicted and some useful research directions towards clarifying and extending the range of situations in which estimates of coverage of fault tolerance mechanisms can be trusted.**

Keywords-software fault tolerance; checker coverage; fault injection; dependability benchmarking

I. INTRODUCTION

Fault tolerance against design flaws is widely recognized to be desirable, in particular in view of the increasing dependence on off-the-shelf software for even critical applications [1].

We address here the problem of assessing the dependability gains achieved by these forms of fault tolerance. There has been a large amount of work on modelling the effectiveness of "software fault tolerance"[1] (see [2] for a review and www.csr.city.ac.uk/diversity for more recent research), to understand what advantages can in theory be expected from it and how these can be pursued through the development process. These models have helped the understanding of the factors that determine the probability of common mode failures between redundant components in these systems. But this research has been mostly limited to "symmetric" architectures: "multiple version software" and similar schemes, in which two or more diverse, redundant components perform equivalent functions and their outputs are "adjudicated" (by voting or some other algorithm, built into the computer system or the physical controlled systems) to decide which values will be output to the controlled system. Such architectures are important for some highly critical applications, but also expensive and relatively rare. In most systems, fault tolerance against design faults takes the form of "asymmetric" architectures, in which a "primary" component (we will often use the term "primary" alone for brevity) performs the required computation, and other components perform error detection, trigger error correction and state recovery mechanisms, or steer the system to a safe state, and so on. These architectures are so commonplace that a list of examples could easily become endless. Asymmetric fault-tolerant schemes are found at all levels of details in designs, from run-time checks within any program or component thereof, to watchdog applications that monitor the whole visible behavior of a complex system for failures, or for safety-relevant failures. Here we are interested in the probability of system failure (that is, a failure of the primary component that the fault-tolerant mechanisms fail to tolerate or mitigate as desired) in these architectures. Despite the huge variation in the details of these systems, we are interested at first in discussing them at a level of abstraction at which they are substantially similar.

We consider the *assessment* of these systems. One can measure reliability of any system by long enough observation of operation or realistic testing. But for critical applications, there is normally a need to predict dependability before a long enough period of observation using reliability models and estimation of component reliability parameters. For symmetric systems, a major difficulty has been found, in that such shortcuts as assuming failure independence between redundant components are not justified [2]. For asymmetric systems, the preferred models rely on *coverage factors*, which could be estimated from field measurements (but with some difficulties) or by *fault injection*.

One of our concerns is that the difficulties in assessing the dependability of symmetric systems are widely recognised, but the methods for assessing asymmetric systems are not usually subjected to the same degree of

[1] Most of the literature refers to software, although fault tolerance against hardware design faults is a recognised need. For instance, both main manufacturers of large fly-by-wire airliners, Airbus and Boeing, use redundant, diverse processors for flight-critical software.

21st International Symposium on Software Reliability Engineering (ISSRE 2010)

scrutiny. We develop here a model to help with this scrutiny, and apply it to practical scenarios with asymmetric systems.

We emphasize that we do not argue for or against any specific architecture, symmetric or asymmetric. Our focus is on highlighting often ignored difficulties in the means for assessing asymmetric systems.

For the sake of simplicity, we will refer our initial discussion of asymmetric systems to a single practical example: *checkers*, i.e., error detection mechanisms (which are common to almost all forms of asymmetric fault tolerance), although the necessary reasoning will be very similar for e.g. error recovery mechanisms.

The rest of the paper is organized as follows. Section 2 discusses primary-checker systems and introduces a basic model and some necessary notations. In section 3 we state the practical problems we would like to address in this paper. In section 4 we present probabilistic models of asymmetric fault tolerance. In section 5 we discuss the implications of the models for the problems stated in section 3. In section 6 we summarize our findings, their limitations and outline directions for future research.

II. PRIMARY-CHECKER SYSTEMS

A. Checkers in software engineering

While N-version programming has been hugely controversial, with both its cost-effectiveness and its effectiveness being questioned, the use of checkers is commonly accepted as self-evidently *good* practice. Many authors in software engineering do not even mention that checkers are a part of fault tolerance, and that knowledge from the fault tolerance literature applies to them, e.g., about the importance of coverage factors[2] (the probability of the checker flagging a failure of the primary, conditional on the failure occurring), and the risk that adding fault-tolerant features to a system may reach a point of diminishing, or even negative, returns. A plausible reason for this general trust in checkers is a belief in simplicity. Checkers only need to verify the results of the primary's computation, not to replicate them. So, they are often simpler than extra versions of the primary would be, and thus cheaper and easier to develop *correctly*. Also, the specification of a checker sometimes has little in common with that of the primary component, making it plausible that the two will not suffer from similar implementation flaws causing common failures. For instance, an important paper about innovative ways of building checkers [3] stated that requiring a checker to be computationally simple is important to achieve this diversity: "we claim (heuristically) that C must be doing something essentially different from what P does, and so, if buggy, may reasonably be expected to make different errors than P. […] we would expect few correlated errors; moreover, we would expect more uncorrelated than correlated errors".

Few would argue against pursuing diversity between primary and checker component; but when it comes to the more quantitative statements quoted, informal judgments like these, when applied to the merits of *symmetric* software fault-tolerance, have at times been found to be misleading. For instance, it seems reasonable to many that independently developed software channels are likely to fail independently in operation, but this has been shown to be unfounded both empirically [4] and theoretically [5]. We will apply to asymmetric software the modeling approach first applied by Eckhardt and Lee [5] to the critique of symmetric fault tolerance, hoping to clarify possible sources of fallacies in intuitive judgments.

The questions to which designers or assessors may need answers include, for instance: how much of a dependability gain would I achieve for my system by adding a particular checker? How effective are checkers of a certain category, for applications in general? What is the effective dependability of a specific system, which uses checkers for fault tolerance?

The problem of assessing the coverage of checker software has been attacked empirically, through fault injection [6], [7] and its application to so-called "dependability benchmarking" [8], [9]. Researchers are aware of the difficulties of selecting representative samples of faults and of extrapolating from measurements that are inevitably based on non-perfectly representative samples [8], [9], [10] , [11]. With respect to these efforts, we wish to define more formally the issues of prediction, and of sampling of faults and errors.

B. Model of asymmetric faut tolerant systems

We choose for analysis the simplest model of an asymmetric fault-tolerant system: a single asymmetric *fault-tolerant component* (FTC) made up of two components, a *primary* and a *checker* (Figure 1). We assume that the operation of the primary and the checker is naturally described in a discrete time frame: the primary performs some operation, and succeeds or fails in it; the checker checks whether the primary succeeded or failed, and it in turn may succeed or fail; how exactly its output is used is not important at this stage. The dependability measures of interest are probabilities of failure per demand (*pfd*).

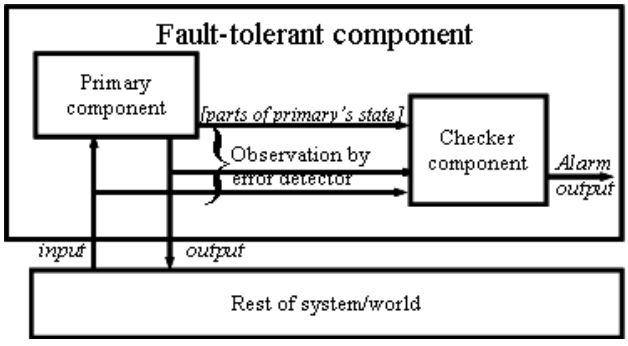

Figure 1. Asymmetric fault-tolerant component (FTC).

[2] We often use the term "coverage" alone for brevity.

C. *Specification of the primary component*

A program's specification describes a *relation* between the spaces of demands[3] and of outputs, i.e. a subset of their Cartesian product: specifically the set of all those {demand, output} pairs formed by a demand value and an output that is correct (according to the specification) for that demand.

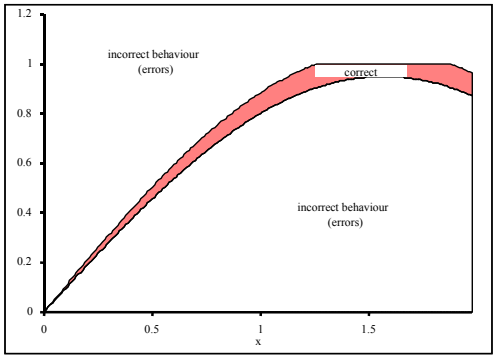

Figure 2. Specification of a single-input, memoryless program as a subset of the Cartesian plane.

For instance, Figure 2 represents, limited to a subset of the {demand, output} space ("demand" to the system – x axis – and "output" of the primary – y axis), a specification requiring a program to calculate $y=sin(x)$, for x in the interval [0,2], with a maximum error of 5% and satisfying the condition $sin(x) \leq 1$[4]. The behavior of a specific program π built to this specification is typically described by a single-valued function (for each demand value x it deterministically produces one output, $O(\pi, x)$), represented in the graph by a set of points, that we would like to lie within the shaded area. If a primary program π produces an erroneous output on an input, then one of these points would lie outside the shaded area in the figure. The projections on the x axis of all such points defines the "failure set" of the program (the set of demands on which it fails).

D. *Specification of the checker component*

A checker typically has as its inputs the outputs of the primary together with the demands submitted to the primary (or parts of each demand: the checker is often built so that it does not have visibility of the internal state of the primary, nor memory of previous inputs to it. Our arguments do not depend on whether the primary has state or whether this is visible to the checker, so we will only refer to our stateless example to illustrate the modeling). The checker's specified output is a Boolean value, where "1" stands for "alarm: the primary's output is erroneous". Its specification is a function whose domain is the set of all possible {demand, output} pairs and whose range is { "OK", "alarm"}.

That is, the checker's specification identifies an "alarm set", within the {demand, output} space, on which the checker is required to issue an "alarm" output. A specific checker σ implements a specific "alarm" function $A_\sigma(\sigma,x,y)$. For instance, in Figure 3 the checker is specified as a logical OR among the violations of several assertions:

$C_\sigma(x,y)=NOT(assertion_1(x,y)) \vee NOT(assertion_2(x,y)) \ldots$

where x indicates the value of the demand, and y the output of the primary. The horizontal stripe at the top in the figure exemplifies the special kind of assertions that check for "illegal" outputs, i.e., outputs that must never be produced, for any demand. A specification may also define "illegal inputs", which are not expected to be used in operation, but may occur, e.g. as a result of accidental failure or malicious activity. In the general case, checkers also check for outputs that are only erroneous for certain demands, but correct for other demands, and thus define more complex patterns, like the triangular wedge on the left in the figure.

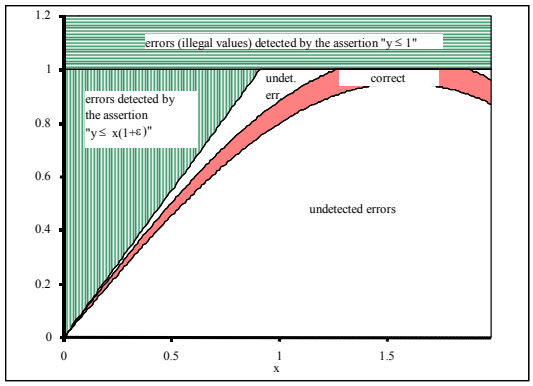

Figure 3. Specification of a possible checker for the primary specified in Figure 2. The areas labelled "errors [..] detected by..." identify input-output pairs for which the specified checker would output "alarm".

On each demand, there are four possible system behaviors: correct behavior of the primary, with no alarm from the checker; *detected failure* (of the primary, flagged by the checker); *undetected failure* of the primary, which we will also call *failure of the fault-tolerant component*; and *false alarm* (the primary's output is correct but flagged as wrong by the checker).

Ideally, we would wish a checker to have 100% coverage, i.e., such that all errors of the primary will trigger an "alarm" output from the checker. The alarm set of such a checker would cover all the space of erroneous {demand, output} pairs, leaving no white space in the Cartesian plane in Figure 2 or 3. We would also usually wish a checker to produce *no false alarms*, i.e., its alarm set to have no overlap with the set of correct {demand, output} pairs. However, either or both goals may be infeasible or uneconomical, and they are often in contrast.

---

[3] We refer to "demands", not inputs, to avoid confusion when referring to stateful systems. "A demand" will mean "the set of values of all input variables and all internal state variables (for a program that preserves state from an invocation to the next) at an invocation of the program".

[4] Specifying just "calculate *sin(x)*" would not make sense with real computer arithmetics: the allowable numerical errors must also be specified. We have allowed errors as large as 5% in this example just so that it can be visualized in the graphs. There may be some subtler requirements, e.g. that the function $O(\pi, x)$ approximates a continuous function and in other ways resemble the graph of a sine function.

It is convenient to identify the class of "*complete checkers*", specified to detect all erroneous outputs from the primary and produce no false alarms, as opposed to "*partial checkers*", specified to miss some failures, for the sake of simplicity, as in Figure 3. For instance, if the primary is specified to compute (exactly) a reversible function [12], a checker that computes the inverse function and compares it against the input to the primary is a complete checker. For a complete checker, the only reason why failures of the primary may go undetected is *imperfect implementation* of the checker. For a partial checker, some "gaps" in the coverage are mandated by the checker's specification. The software engineering literature tends to deal with checkers specified to produce no false alarms; by contrast, checkers for safety and security may well be specified to produce even frequent false alarms for the sake of reducing the risk of undetected failures. In this paper, for reasons of space, we do not consider false alarms and the attendant design trade-offs.

E. The uncertainties affecting assessment

The benefits brought by a checker are naturally quantified through a coverage factor, i.e., the probability of it correctly flagging an incorrect output of the primary, on a randomly sampled demand to the system. This probability summarises the effects of three sources of uncertainty (Figure 4): which faults are present in the primary component, which faults are present in the checker component, and the value of the demand itself (and thus whether it will trigger one of the faults in the primary, and in such a way that the resulting error will or will not be flagged by the checker).

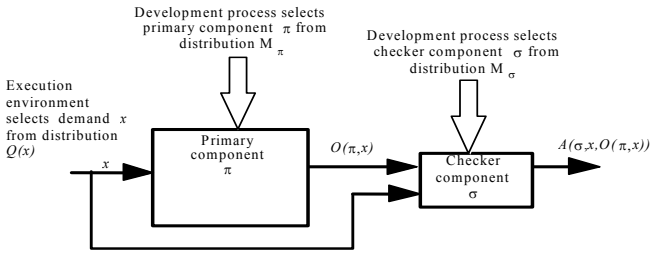

Figure 4. Sources of uncertainty affecting whether a demand will cause failure of the primary component and whether this will be detected by the checker component.

An elegant representation for uncertainty about faults is as follows [5]. Design faults, for instance in the primary, are created by the software development process. The development process can be seen as a process of random selection from a probabilistic distribution of *all the possible primary components* which could be created to the given specification. Each possible primary, π, is described by its demand-to-output mapping, $O(\pi,x)$. We will designate the primary developed through this random selection by a random variable Π; for a specific primary, e.g., π, saying that we do not know its faults means that we do not completely know the function $O(\pi,x)$ that π actually implements[5]. The same ideas apply to the faults of the checker. The creation of physical faults can also be seen as sampling, according to a specific probability distribution, from a population of possible components, which includes the non-faulty component, say π^*, and all possible faulty ones π_i, characterised by having different behaviours $O(\pi_i,x)$.

Given a population $\wp_\pi$ of the possible primaries, we will call $M_\pi(\pi)$ the probability of the particular primary π being chosen at random from $\wp_\pi$ by the development process. This probability distribution is normally unknown; the purpose of this notation is to highlight the effects of this form of "randomness" on the dependability measures of interest. Similarly, we can designate a probability $M_\sigma(\sigma)$ of a specific checker σ being selected by the development process from the population of the possible checkers, $\wp_\sigma$. Last, the value of the demand is selected, by the environment in which our FTC operates, according to a probability distribution $Q(x)$.

In this paper, we will consider a restricted set of problems in which, given a specific checker, one wishes to assess the dependability gains that it would bring to a specific primary, or to a primary obtained from a given population $\wp_\pi$ and distribution $M_\pi(\pi)$.

It is worth pointing out that once a primary and a checker components have been assembled into a system (our "FTC"), the demand profile for the checker (i.e. the probability distribution of the $<x, y>$ pairs submitted to it) depends on the primary. For a particular demand, x, and a particular primary, π_1, the checker will see the pair $<x, O(\pi_1, x)>$. Given a different primary, π_2, on the same demand, x, the checker would see a pair, $<x, O(\pi_2, x)>$, which may be different. In both cases the demand profile for the primaries is the same, $Q(x)$, but the profiles for the checker will differ, having in common that the marginal distribution of the variable x, $Q(x)$, is the same.

III. PROBLEM STATEMENT

We are interested in assessing the effectiveness of fault tolerance in asymmetric fault-tolerant systems, with a specific interest in (a) checkers as a concrete category of fault tolerance mechanisms and (b) fault tolerance against software (or generally design) faults. We consider the problem of assessing coverage factors experimentally, and the use of fault injection to this end. There is a range of scenarios of application of this approach to assessment, and to highlight the issues that affect its validity we will identify two concrete scenarios to represent opposite extremes of the range:

[5] We will use uppercase letters to designate random variables, and lowercase letters to designate specific values. Note that if an argument of a function is a random variable, the value of the function becomes itself a random variable.

1. estimating coverage with respect to transient faults caused by cosmic ray particles in a particular computer system (i.e. physical faults rather than design faults). For the estimation, faults are produced by fault injection. A specific primary software component is the target of the measurement, and the fault injection creates a population of alternative primaries. The failures of the primary are counted, and so are the successful detections of these failures by the checker. The ratio of the two is an estimate of coverage.
2. "dependability benchmarking" with respect to software faults. This is a variation of the previous scenario. The purpose is typically to compare different products of similar functionality by subjecting each one to the same procedure of fault injection. The products can be seen as fault-tolerant components (in our terminology), and the observed effects of each injected fault can be classified as tolerated or not tolerated (leading to system failure), so that a coverage factor can be estimated. The goal is often to rank the "benchmarked" products according to how well they coped with the set of injected faults.

In terms of our model, in both cases, fault injection (or the injection of errors in the state of the target systems to emulate the effects of faults) creates a distribution of alternative primary components. We will artificially simplify the models by assuming that checkers are not affected by fault injection and do not change between measurement and operation.

We will look at these problems more formally and scrutinize the rationale behind the current and proposed practices for dealing with them.

IV. MODEL OF ASYMMETRIC FAULT-TOLERANT SOFTWARE

A. Score functions

We start by modeling the behavior of the primary and the checker[6]. We first *specify* the primary via a function $O(x)$, whose value for each demand x is the set of acceptable outputs. The output value that an implementation of the primary produces on demand x must belong to the set $O(x)$. In the example in Figure 2, $O(x)$ is represented in the graph by the set of y values shown in grey for the given x.

Let us denote as $O(\pi,x)$ the value that a particular primary, π, produces in response to demand x. $O(\pi,x)$ will be an element of the set $O(x)$ iff the primary processes the demand correctly and will be outside the set $O(x)$ iff the primary fails.

[6] This modeling is conceptually the same as that used in [13], which relies on defining the coverage conditional on a certain {activity, fault} pair. However, we describe the set of possible events with very fine granularity, to emphasize that the checker's coverage, which is a probability conditional on a class of events, is normally an average of a 'pointwise' coverage that is deterministically 0 or 1 for each {demand, fault} pair.

Now let us define the *score function* $\omega(\pi,x)$ for a particular implementation, π, of the primary as the following indicator function:

$$\omega(\pi,x)=\begin{cases}0, if\ O(\pi,x)\in O(x),\\ 1, elsewhere\end{cases} \quad (1)$$

Its expected value, $\theta(x)$, over $\wp_\pi$ will give the probability that a randomly chosen primary will fail when processing demand x. Following Littlewood and Miller [14] we call this expected value *difficulty* of demand x.

The value, $O(\pi,x)$, which the primary produces when processing demand x varies between implementations of the primary. We capture this variability using another (indicator) function, $l_\pi(\pi,x,y)$:

$$l_\pi(\pi,x,y)=\begin{cases}1, if\ O(\pi,x)=y\\ 0, elsewhere\end{cases} \quad (2)$$

If there is a population of primaries, for instance created by faults, the expected value, $L_\wp(x,y)$, that this function takes over the population of primaries, $\wp_\pi$, represents the probability that a randomly chosen primary, given the demand x, will produce y, and thus presents the checker with the pair (x,y).

$$L_\wp(x,y)=\sum_\wp l_\pi(\pi,x,y)M_\pi(\pi) \quad (3)$$

Note that $L_\wp(x,y)$ is a conditional probability (of the "randomly chosen" primary producing output y given demand x) and indeed, $\sum_y L_\wp(x,y)=1$.

The function $L_\wp(x,\bullet)$ is important because it defines the *demand profile* created for the checker by the primaries (when processing each specific demand x). It depends on the population $\wp_\pi$ and the distribution $M_\pi(\bullet)$ defined on $\wp_\pi$. That is, any change in the process that produces $M_\pi(\bullet)$ and $\wp_\pi$ – the software development process, or a physical fault process – may change $L_\wp(x,\bullet)$.

Now we turn our attention to the checker. We define a "score function" for checker σ as the indicator function:

$$\omega_\sigma(\sigma,x,y)=\begin{cases}0, if\ either\ y\in O(x)\ or\\ \sigma\ detects\ correctly\ that\ y\notin O(x).\\ 1, elsewhere\end{cases} \quad (4)$$

Note that we have defined this score function to flag the *false negative* failures of the checker (i.e. when the checker fails to detect a failure of the primary) as 1, but not to flag the *false positives* (false alarms), because we are not going to analyze the probability of false alarms. The definition of the score function is further illustrated by Figure 5.

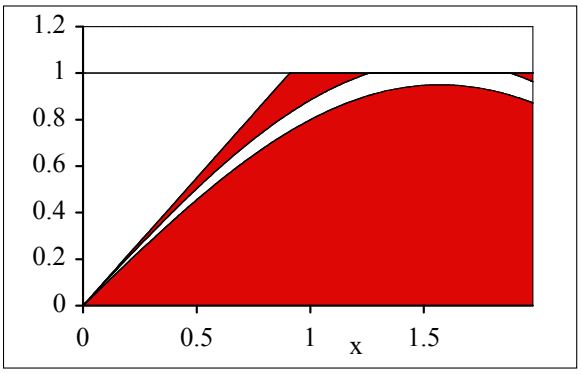

Figure 5. The ω_σ function for the checker specified in Figure 3, if implemented correctly. $\omega_\sigma(\sigma,x,y)$ = 0 in the white area, and $\omega_\sigma(\sigma,x,y)$ = 1 in the grey areas.

B. *Probability of undetected failure and coverage*

Given a specific primary π, checker σ and demand x the following 3 combinations are possible for the pair of scores [$\omega(\pi,x)$, $\omega_\sigma(\sigma,x,y)$]:

- 00: the primary processes x correctly, and the checker outputs 'OK' or raises a false alarm;
- 10: the checker detects a failure of the primary;
- 11: the checker misses a failure of the primary (i.e. the fault-tolerant component fails).

The fourth combination, 01 is impossible due to our definition (4).

Each of these outcomes will have a probability (for a randomly chosen primary and a given checker). We will denote these as p_{00}, p_{10} and p_{11}, respectively (omitting the argument (x) for brevity). For the "difficulty" of the demand for the primary, $\theta(x)$, defined earlier, the following holds: $\theta(x) = p_{10} + p_{11} = 1 - p_{00}$.

1) Probability of undetected failure

Using the indicator functions and probabilities, defined so far, we can express directly the probability of undetected failure as follows:

$$P(\textit{undetected failure}) = \sum_x \left(\sum_{\pi,y} \omega(\pi,x) I(\pi,x,y) \omega_\sigma(\sigma,x,O(\pi,x)) M_\pi(\pi) \right) Q(x) = \sum_x \left(\sum_y \omega_\sigma(\sigma,x,y) L(x,y) \right) Q(x). \quad (5)$$

where the inner summation represents the probability of undetected failure given a specific demand x, and the outer summation averages this over the demands. The probability of failure of the primary (on average over our notional "population" with different faults) is:

$$P(\textit{failure of primary}) \equiv P_{fp} = \sum_x \left(\sum_\pi \omega(\pi,x) M_\pi(\pi) \right) Q(x) = \sum_x \theta(x) Q(x) = \mathrm{E}_Q[\theta(X)]. \quad (6)$$

where E designates the expected value.

2) Checker coverage

The coverage of the checker is then expressed using the ratio of the two expressions above:

$$\textit{Coverage} = 1 - \frac{\sum_x \left(\sum_y \omega_\sigma(\sigma,x,y) L(x,y) \right) Q(x)}{\sum_x \theta(x) Q(x)}. \quad (7)$$

This equation highlights how the coverage depends both on the frequencies of different demands, represented by $Q(x)$, and on the frequency with which primaries produce the various possible outputs, represented by $L(x,y)$.

3) Variation of coverage over the demands

One can actually define a coverage factor for each specific demand x, $C_\sigma(\sigma,x)$, and using it allows some useful observations. The probability of failure of the checker to detect a failure of a primary on a specific demand x, i.e. directly the probability of undetected failure of the fault-tolerant component on x, is:

$$\mathrm{E}_y(\Omega_\sigma(\sigma,x,Y)) = \sum_y \omega_\sigma(\sigma,x,y) L(x,y) \quad (8)$$

that is, the expected value of the score of the checker with respect to the measure $L_{\wp}(x,\bullet)$. We can also state:

$$C_\sigma(\sigma,x) \equiv P(\textit{failure flagged} \mid \textit{primary failed on } x) = \frac{P(\textit{failure flagged}, \textit{primary failed on } x)}{P(\textit{primary failed on } x)} = \frac{p_{10}}{p_{11}+p_{10}} = \frac{p_{10}}{1-p_{00}}. \quad (9)$$

From here we can express:

$p_{10} = C_\sigma(\sigma,x)(1-p_{00})$, which trivially leads to expressing the probability of interest, p_{11}, that the checker will miss a failure of the primary on x as follows:

$$p_{11} = 1 - p_{00} - p_{10} = 1 - p_{00} - C_\sigma(\sigma,x)(1-p_{00}) = (1-p_{00})(1-C_\sigma(\sigma,x)) = \theta(x)(1-C_\sigma(\sigma,x)) \quad (10)$$

The expression above again highlights that the coverage is a function of the profile, $L_{\wp}(x,\bullet)$, generated by the population of primaries. As discussed earlier, any change in the software development process, or a physical fault process, may thus change the coverage.

All this was derived for a specific demand, x. The probability of system failure on a randomly chosen demand, X, for specific demand profile, $Q(\bullet)$, becomes:

$$\sum_x \theta(x)(1 - C_\sigma(\sigma, x))Q(x) = \\ \mathrm{E}_Q[\theta(X)] - \mathrm{E}_Q[\theta(X)]\mathrm{E}_Q[C_\sigma(\sigma, X)] - \\ covariance_Q(\theta(X), C_\sigma(\sigma, X)) = \\ P_{fp}(1 - Coverage_\sigma) - covariance_Q(\theta(X), C_\sigma(\sigma, X)). \quad (11)$$

where $Coverage_\sigma \equiv \mathrm{E}_Q[C_\sigma(\sigma, X)]$ is the expected value (over the demand space of the primaries) of the checker's coverage with respect to the failures of a randomly chosen primary conditional on the individual demands. Note that $Coverage_\sigma$ is *different* from *Coverage* in equation (7), the true coverage of the checker with respect to the failures of a randomly chosen primary.

This highlights a possible error in estimating the true coverage. For the purpose of testing and measurement, it is common to partition the demand space into non-overlapping subsets (thought to be equivalence classes). One may be tempted to estimate the checker coverage in each partition first, and then compute the sum of these estimates, weighted with the probabilities of demands from each partition. However, this process estimates $Coverage_\sigma$, not *Coverage*. As shown by formula (11), multiplying $Coverage_\sigma$ by the estimated marginal probability of failure of the primary would not yield the probability of detected failure, due to the covariance term in (11).

The sign of the covariance term is unknown, hence ignoring it may lead to an error with unknown sign (under- or overestimation): this method does not even yield a useful bound (either conservative or optimistic).

V. DISCUSSION

We now discuss important, negative implications of the models, and will then proceed to discuss the scope of these negative implications, and what new knowledge would be useful for reducing it.

A. Effectiveness of checkers vs "symmetric" fault tolerance

We started by considering how the use of checkers against software faults (asymmetric fault tolerance) tends to be less controversial than that of multiple-version (symmetric) fault tolerance. Depending on the system and project, the asymmetric solution may offer lower costs and/or higher coverage; whether this is true in a specific scenario can only be decided by empirical evidence. But for the purpose of *assessing* the achieved dependability, asymmetric solutions bring no advantage.

A first observation about equation (11) is that its summation-of-products form is similar to those describing the probability of failure of a two-version, 1-out-of-2 system [2] and it carries the same message that:

- not only does the system *pfd* depend on how the system is used (the demand profile), and thus empirical estimates of reliability obtained under different profiles cannot be trusted
- but assessing each subsystem in isolation (the *pfd*s of the two versions, or the *pfd* of the primary plus coverage of the checker), even using the correct demand profile, does not seriously help to estimate the probability of system failure, due to the covariance term in (11). Actually, the *coverage of the checker cannot even be defined* without assuming a primary or a distribution of primaries.

An aspect of this for symmetric systems is that if a change in one of the two versions improves its *pfd*, this does not guarantee improvement of the system *pfd*, but might even make it worse, if the increase in version reliability has reduced diversity between the two: the improved version has more failure points that coincide with failure points of the other version. Likewise, in an asymmetric system, improving the *pfd* of the primary does not necessarily improve the probability of undetected failure. The only component improvements that are guaranteed a priori to improve the probability of system failure are those that make a component correct on some demand on which it previously failed, but do not cause it to fail on any demand on which it was previously correct. An example of component improvement that may make the probability of system failure worse is a fix of a bug in the primary – which caused failures detected by the checker – which leads to introducing another bug, against which instead the checker is ineffective.

B. Experimental estimation

We wish especially to discuss to what extent one can depend on estimates of coverage, or of probability of undetected failure, obtained experimentally via fault injection.

At this point, we note that none of the discussion that follows is actually specific to error detection: the issues concern the prediction of the probability of *any* fault tolerance mechanism failing to tolerate some fault or error condition, conditional on the latter occurring (prediction of coverage factors) or unconditionally (prediction of probability of certain system failures).

In the previous section, we started with recognizing that whether an undetected error occurred would depend on how an injected fault altered the primary component (turning it into another primary within our "population" of possible primaries) and on which demand was submitted to it. This shows how both the distributions of the demands and of the faults affect the resulting probability of undetected failure and thus coverage of the fault-tolerant mechanisms; on the other hand, it is not necessarily a convenient way of estimating the measures of interest.

In an experimental approach to estimating these quantities, one samples the distributions of demands and faults by applying what are often called a "work load" and a "fault load". What exactly these terms mean depends on which kind of system and fault-tolerant mechanisms are being measured. For instance, the system under consideration could be a hardware platform, the fault load a mechanism for producing hardware faults and the work load a certain software configuration with a mechanism for generating demands to it. The fault load is typically produced through some form of fault injection, so as to

reduce component reliability and compress the amount of time required (to observe enough faults to get narrow enough confidence intervals for coverage estimations). If these coverage estimates could be extrapolated to different environments, the rate of system failure in the latter would be just the estimated coverage (specific to the fault tolerance mechanisms) times the rate of occurrence of faults in the specific environment.

However, this extrapolation is generally *invalid*. As our equations (7, 10, 11) show, if the fault load and/or work load change – for instance, the system just outlined is moved to an operational environment with a different distribution of demands, or used with different software, thus changing *Q(x)* and/or $M_\pi(\pi)$ – all the measures of interest can in principle change.

Extrapolating estimates to a new environment requires judgment about how the new environment changes the terms in equation (11), which may be difficult. Simply assuming that the coverage will remain unchanged in the new environment (an implicit assumption made whenever a coverage factor is considered as an intrinsic feature of a fault-tolerant mechanism) is a very strong assumption. It assumes that in equations (10) and (11), despite changes in the weights used in the various weighted averages, averages or ratios of averages remain the same.

In principle, instead, a change in either the fault load (the distribution of primaries, in our model) or the demand profile may change the coverage to any value between 0 and 1. These two extreme values are taken if the only {demand, fault} pairs possible in the new environment are, unfortunately, pairs that produce errors that are not covered (for which $\omega_\sigma(\sigma, x, y)$ function is 1), or, fortunately, pairs that are covered ($\omega_\sigma(\sigma, x, y)$=0). These extreme cases seem unlikely, but they make it useless to have an estimate of coverage in an artificial environment unless one can build some argument about the likely magnitude of the errors made by extrapolating the estimate to a different environment. But these arguments are usually *absent* in reports of experimental assessment of coverage.

All that precedes does not mean that estimates obtained experimentally are always completely untrustworthy for a different environment. To discuss the factors that should enter a judgment about extrapolability, we discuss two extreme scenarios as outlined in section III.

C. *Transient faults due to radiation*

In this scenario, the target system is a complete computer (hardware and software configuration) and we are interested in its behavior when affected by radiation (cosmic rays) . We can informally think of the life of the fault-tolerant target system (or of multiple, identical copies of it) as a sequence of time frames. In some of these, the primary is fault-free; in others, it has been turned by a particle impact into one of a "population" of possible "mutant" primaries. Over time, the probabilities of these various alternative scenarios define a probability distribution for the population of possible primaries.

A specific checker is used with this population of primaries. (If desired, one can also consider the possibility of faults changing the checker. We have omitted this possibility in section IV for reasons of space.)

Fault injection can take the form of exposing the primary to more intense radiation than expected in real operation, but with the same spatial distribution and the same mix of radiation types. This increases the relative frequency of the time frames in which a fault occurs, but there is no reason for suspecting that the distribution of faults covered or not covered by checkers will also change.

In this ideal scenario, it *seems right to expect* that a coverage estimate obtained with fault injection will be accurate enough for predictions about real operation. There are bounds to the validity of this argument; for instance, if the artificially intense radiation increases the frequency with which further faults occur before the fault tolerance mechanisms have finished responding to a previous fault, this would change the distribution of faults (that is, of "possible primaries"). However, one can assess whether a substantial estimation error is likely by measuring the fault frequency and response times in question

D. *Dependability benchmarking for software faults*

Research into dependability benchmarking is inspired by the goal of achieving objective methods for *assessing aspects of software quality*, or at least for *ranking competing products or solution* (with similar functionality) from the viewpoint of dependability or of robustness (that is, of coverage of their defensive mechanisms against the expected faults, or more generally threats or disturbances) [15]. This goal is clearly desirable, but we now scrutinize its feasibility.

In the ideal scenario, a set of competing software products (workloads) could be "benchmarked" by experiments of fault/error injection. The experiments apply the same "fault load", as far as possible. For instance, a set of common types of software faults is identified and then injected into the run-time images of the various software products to be compared [11]. Statistics are collected on how well the different workloads cope with the same fault load, and used to rank the products.

The measured frequency (estimate of probability) of system failure for each population of mutants derived from the same product is used to rank the competing products. If the estimates of failure probability are correct, given the same fault load they are also proportional to the coverage factors for the various products.

The fault load may address different concerns. For instance [16] it might include operator mistakes (e.g. deleting individual files, dropping tables from a databases, etc.), software faults, hardware faults, etc. We focus here on software faults.

Mapping this scenario on the model of section IV, each one of the products being assessed is a single primary-checker pair (that is, the distribution of primaries is a degenerate one with a single primary). The "checker" is the composition of all the mechanisms that contribute to tolerating errors. The fault injection experiment substitutes

the degenerate distribution – the single, real primary – with a different distribution, in which, instead of the real product, "mutant" ones are possible, each one having the faults of the real product, plus one fault injected from the chosen fault load.

Apart from increasing the primary's probability of failure, the effect of fault injection on the parameters in equation (7) and (11) is difficult to foresee. The probability of an injected fault causing an undetected (system) failure (or of it being instead "covered") may be totally unrelated to the same probability for any one of the true faults. Furthermore, the error between the experimental estimate and the true value of coverage may differ between the products being evaluated, in unpredictable ways, since each product has its own specific faults, which may or may not resemble what the experimenter considers a representative sample. If via "benchmarking" one tries to compare products A, B, C, ..., it is far from obvious whether the ranking produced, based on the mutants of A, the mutants of B, etc, will be the same as the correct ranking that could be obtained among A, B, C, ..., e.g. by long operational testing or long operational exposure. This concern applies both to statements about dependability and statements about coverage.

There is *no a priori reason* why the best product cannot end up with the worst population of mutants and vice versa. This *problem seems inherent to dependability benchmarking* targeting design (software) faults.

Fault injection is certainly useful for debugging fault tolerance mechanisms. But in terms of *assessment*, all that can be certainly said is that injecting a standard set of faults can be a simple test of development competence: an assessor may think that defensive design is an important aspect of quality, and thus any product that does not tolerate a certain subset of injected faults should be considered inadequate. The "dependability benchmark" would then be used as a true "benchmark" in its older meaning, of a single reference point rather than a measurement protocol. But in terms of ranking of products, there is a dearth of convincing arguments to demonstrate that the ranking between the real products will carry through fault/error injection, and thus the *trust in dependability benchmarking seems excessive*.

VI. CONCLUSIONS AND FUTURE RESEARCH

We have discussed the difficulties in assessing the effectiveness of "asymmetric" schemes, e.g. of the primary-checker type, for tolerating design faults, and we have presented a model to assist with this analysis. We used a style of conceptual modeling similar to that previously applied to symmetric systems (e.g., N-version programming) [2], which focuses on the deterministic success or failure of the fault tolerance mechanisms with respect to each specific fault-demand combination.

This modeling is not substantially new, but helps to illustrate important points that are often neglected in current literature.

About the general problem of assessing "asymmetric" fault tolerance, like defensive programming, watchdog components, etc, ***we have pointed out that it is no less difficult than for "symmetric" fault tolerance***; ***in neither case can one evaluate the redundant components in isolation*** and then combine measures thus obtained to obtain a measure for system-level dependability. Indeed, the very concept of *measuring a checker in isolation is suspect or meaningless*, since any coverage measure depends on the distribution of "demands'" coming to the checker from the checked ("primary") component. The coverage of the checker, which is often attributed to the checker alone, is in fact a characteristic of the checker and the particular primary (for design faults, present in the primary), i.e. the checker performance may (and typically will) vary with the primary. This observation is not particularly new. The model presented, however, shows in detail the mechanisms of this dependence and thus the *fallacies of arguments* in which this dependence is ignored.

We emphasize again that we are not advocating symmetric systems over asymmetric systems, or vice versa. The choice of a fault-tolerant architecture must take into account many factors. We have, however, pointed out that the difficulty of assessing asymmetric systems is often underestimated.

We have then turned to methods for assessing coverage factors, and argued that the goal of assessing them via fault injection and dependability benchmarking may be unfeasible. Trusting these assessment methods too often ignores their recognized limits as *dependability assessment* tools (as opposed to the obvious usefulness of fault injection for *achieving* dependability, by debugging fault tolerance mechanisms). "Without knowledge about [the probability distribution of faults and activities] one cannot make any meaningful statement about the coverage factor of a system" [13]. The problem is simply the degree to which measurements in one environment can be extrapolated to others.

A gap in current practice is highlighted by the fact that many papers reporting fault injection results acknowledge that the validity of the results hinges on the appropriateness of the "fault load" and "work load" used, but decline to discuss the degree of general usefulness of the measures obtained: which results will probably be useful predictions, in what range of situations, with what expected error - and why? But surely this is what matters. Real operation will always be different from the experimental set-up, but if this implied that the measures were of no consequence outside the lab where they were obtained, why report them at all? The important point here is that any extrapolation of the measures to different environments, any claim on how small the resulting error should be expected to be, should be justified by arguments specific to the scenario in question, rather than trusted *a priori*. We have pointed out that there is a range of scenarios, from ones in which extrapolation seems obviously correct, to ones in which it seems completely untrustworthy. The latter include the scenario of interest for this paper: assessing the coverage factor of means for detecting or tolerating software design faults.

While frequently acknowledging in general terms the limitations to the validity of the measures obtained, researchers and practitioners seem often to adopt, without

explicit justification, two simplificatory heuristics:

- injecting "representative sets" of faults, in the sense of faults that appear to be reasonably common in practice, without regard to whether they are a representative *sample* of the true probability *distribution* that will occur in real operation;
- mistrusting measures obtained with fault injection as estimates of the corresponding "true" measures in operation, but trusting them as valid indications of *ranking* between the "true" measures.

It seems thus important to have repeated that neither is correct in general, and whether predictions thus obtained should be trusted, and which magnitude of error should be expected, ought to be justified by explicit arguments, based on knowledge of the system and fault mechanisms and/or statistical evidence.

There is certainly scope for useful research (some directions are discussed in [17], Appendix A1). First, the community needs predictions obtained from fault injection (about values of coverage factors, reliability measures) to be cross-checked against the corresponding measures in real operation. Publishing this kind of results would help to refute or corroborate the trustworthiness of these assessment methods. It is normal in engineering that methods based on wrong assumptions turn out to be accurate enough at least for some classes of problems. But only experimental evidence can indicate the area of validity of these "theoretically wrong" methods. Research comparing empirically the measures produced by different fault injection methods [18-20] is also an indirect contribution in this direction.

A second kind of research that is needed concerns making explicit the reasoning, evidence and hidden assumptions behind trust in "theoretically wrong" coverage estimation methods, or ranking methods. We have cited the example of fault injection by intensified radiation as one in which the validity of the method can be demonstrated by reasoning about the physics of the fault process. In other cases in which estimation by fault injection is indeed appropriate, an argument for believing it to be so may be much more complex. It might involve experimental evidence about types of faults, their frequency and effects, as well as deductive reasoning from design characteristics. It may rely on bounds on the terms of our equations, on evidence for the appropriateness of specific stratification methods for the fault injection, etcetera.

Methods for structuring complex arguments ("cases") have proven very useful in an increasing range of areas requiring complex judgments ("safety cases", "assurance cases") [21] and may prove useful here to decide in which situations trust in fault injection is misplaced, or can be justified, or could be justified if specific additional evidence were produced.

ACKNOWLEDGMENT

This work has been partially supported by the following projects: DOTS (Diversity with Off-the-Shelf Software), and INDEED (INterdisciplinary DEsign and Evaluation of Dependability) both funded by the U.K. Engineering and Physical Sciences Research Council, grant GR/N23912 and EP/E001580/1, respectively, and FP7 Coordination Action AMBER (Assessing, Measuring and Benchmarking Resilience), funded by the European Commission (FP7 - Project 216295).

REFERENCES

[1] Popov, P., L. Strigini, and A. Romanovsky. Diversity for off-the-Shelf Components, in DSN 2000 - Fast Abstracts supplement. 2000. New York, NY, USA: IEEE Computer Society Press.

[2] Littlewood, B., P. Popov, and L. Strigini, *Modelling software design diversity - a review,* ACM Computing Surveys, 2001. **33**(2): p. 177-208.

[3] Wasserman, H. and M. Blum, *Software Reliability via Run-Time Result-Checking,* Journal of the ACM, 1997. **44**(6): p. 826-849.

[4] Knight, J.C. and N.G. Leveson, *An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming,* IEEE Transactions on Software Engineering, 1986. **SE-12**(1): p. 96-109.

[5] Eckhardt, D.E. and L.D. Lee, *A theoretical basis for the analysis of multiversion software subject to coincident errors,* IEEE Transactions on Software Engineering, 1985. **SE-11**(12): p. 1511-1517.

[6] Arlat, J., et al., *Fault Injection for Dependability Validation - A Methodology and Some Applications,* IEEE Transactions on Software Engineering, 1990. **16**(2): p. 166-182.

[7] Hsueh, M.-C., T.K.a. Tsai, and R.K. Iyer, *Fault Injection Techniques and Tools,* Computer, 1997. **30**(4): p. 75-82.

[8] Kanoun, K., et al. *DBench Dependability Benchmarks*, 2004 [cited 11 September 2009]; http://www.laas.fr/DBench/.

[9] Kanoun, K. and L. Spainhower, eds. *Dependability Benchmarking for Computer Systems*. 2008, Wiley - IEEE Compute Society Press Hoboken, New Jersey. 362 p.

[10] Durães, J. and H. Madeira, *Emulation of Software Faults: A Field Data Study and a Practical Approach,* IEEE Transactions on Software Engineering, 2006. **32**(11): p. 849-867.

[11] Durães, J. and H. Madeira. Definition of Software Fault Emulation Operators: A Field Data Study, in DSN - 2003. San Francisco, CA, USA: IEEE Computer Society.

[12] Bishop, P.G. Using reversible computing to achieve fail-safety, in ISSRE'97 Int. Simposium on Software Reliability Engineering. 1997.

[13] Cukier, M., D. Powell, and J. Arlat, *Coverage Estimation Methods for Stratified Fault-Injection,* IEEE TC, 1999. **48**(7): p. 707-723.

[14] Littlewood, B. and D.R. Miller, *Conceptual Modelling of Coincident Failures in Multi-Version Software,* IEEE Transactions on Software Engineering, 1989. **SE-15**(12): p. 1596-1614.

[15] Kanoun, K., et al., Prologue: Dependability Benchmarking: A Reality or a Dream, in [9], p. xiii-xviii.

[16] Viera, M., J. Durães, and H. Madeira, *Dependability Benchmarking for OLTP Systems*, in [9], p. 63-90.

[17] Bondavalli, A. and P. Lollini, eds. *Assessing, Measuring, and Benchmarking Resilience ," Final Research Roadmap"*. 2009, EU FP7 project No 216295 , AMBER:Assessing, Measuring, and Benchmarking Resilience p.

[18] Arlat, J., et al., *Comparison of Physical and Software-Implemented Fault Injection Techniques,* IEEE Transactions on Computers, 2003. **52**(9): p. 1115-1133.

[19] Jarboui, T., et al. Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study, in Pacific Rim International Symposium on Dependable Computing. 2002: IEEE Computer Society.

[20] Christmansson, J., M. Hiller, and , and M. Rimen. An Experimental Comparison of Fault and Error Injection, in 9th International Symposium on Software Reliability Engineering (ISSRE-9) 1998.

[21] Bloomfield, R.E., et al., *International Working Group on Assurance Cases (for Security),* IEEE Security & Privacy, 2006. **4**(3): p. 66-68.